



US011436030B2

(12) **United States Patent**
Hulick, Jr.

(10) **Patent No.:** **US 11,436,030 B2**
(45) **Date of Patent:** **Sep. 6, 2022**

(54) **MODULAR JAVA MANAGER PERFORMING CAPTURING AND OVERRIDING OF ACCESS CHECK FAILURES FOR INTER-MODULE OPERATIONS BETWEEN JAVA MODULES BY AN AGENT USING INSERTED INSTRUMENTATION**

(71) Applicant: **Cisco Technology, Inc.**, San Jose, CA (US)

(72) Inventor: **Walter Theodore Hulick, Jr.**, Pearland, TX (US)

(73) Assignee: **Cisco Technology, Inc.**, San Jose, CA (US)

(*) Notice: Subject to any disclaimer, the term of this patent is extended or adjusted under 35 U.S.C. 154(b) by 205 days.

(21) Appl. No.: **16/788,041**

(22) Filed: **Feb. 11, 2020**

(65) **Prior Publication Data**
US 2021/0247992 A1 Aug. 12, 2021

(51) **Int. Cl.**
G06F 9/455 (2018.01)
G06F 9/445 (2018.01)

(52) **U.S. Cl.**
CPC **G06F 9/45529** (2013.01); **G06F 9/44589** (2013.01); **G06F 9/45558** (2013.01); **G06F 2009/45583** (2013.01); **G06F 2009/45595** (2013.01)

(58) **Field of Classification Search**
CPC **G06F 9/45529**; **G06F 9/44589**; **G06F 9/45558**; **G06F 2009/45583**; **G06F 2009/45595**

See application file for complete search history.

(56) **References Cited**

U.S. PATENT DOCUMENTS

7,668,953	B1 *	2/2010	Sinclair	H04L 41/16 709/224
2007/0180439	A1 *	8/2007	Sundararajan	G06F 11/3644 717/158
2007/0233869	A1 *	10/2007	Jodh	H04L 63/10 709/226
2011/0078388	A1 *	3/2011	Gyuris	G06F 12/1027 711/E12.001
2015/0178057	A1 *	6/2015	Miadowicz	G06F 9/4552 717/151
2018/0268158	A1 *	9/2018	Bateman	G06F 21/6218
2021/0247966	A1 *	8/2021	Hulick, Jr.	G06F 11/3409

OTHER PUBLICATIONS

Deitel, Paul, "Understanding Java 9 Modules", online: <https://www.oracle.com/corporate/features/understanding-java-9-modules.html>, Sep. 2017, 9 pages, Oracle.com.

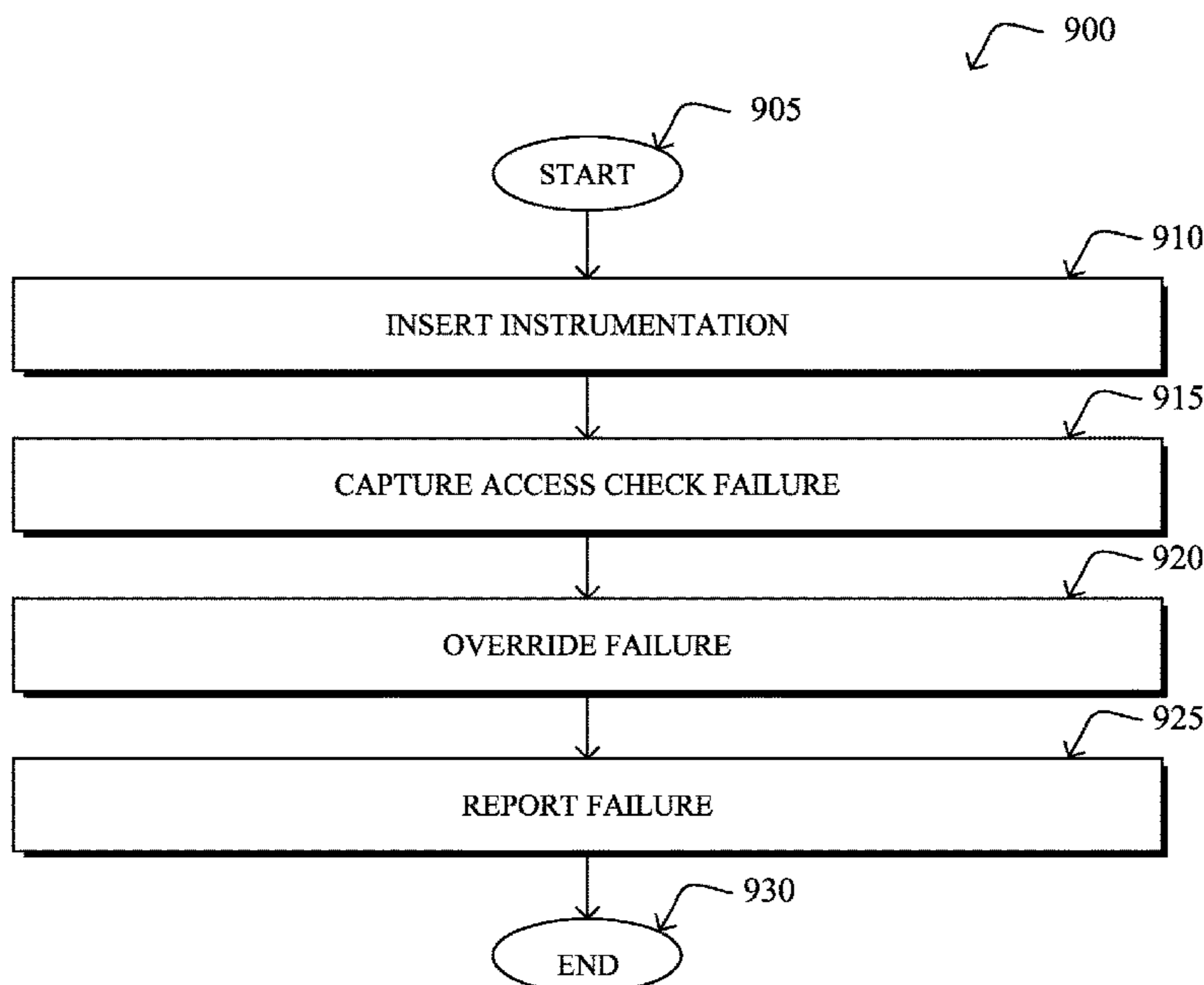
(Continued)

Primary Examiner — Michael W Ayers
(74) *Attorney, Agent, or Firm* — Behmke Innovation Group LLC; James M. Behmke; Jonathon P. Western

(57) **ABSTRACT**

In one embodiment, an agent inserts instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed. The agent captures, using the instrumentation, an access check failure for an inter-module operation between the Java modules. The agent overrides, using the instrumentation, the access check failure. The agent reports the captured access check failure to a user interface.

20 Claims, 16 Drawing Sheets



(56)

References Cited

OTHER PUBLICATIONS

Parlog, Nicolai, “Five Command LineOptions to Hack the Java Module System”, online: <https://blog.codefx.org/java/five-command-line-options-hack-java-module-system/>, 2017, 16 pages, Blog.CodeFX.org.

Sheasha, Dalia Abo, “Migration to Java 11 Made Easy”, online: <https://developer.ibm.com/tutorials/migration-to-java-11-made-easy/>, Feb. 26, 2019, 8 pages, IBM.com.

“Class Module (Java SE 9 & JDK 9)”, online: <https://docs.oracle.com/javase/9/docs/api/java/lang/Module.html>, Sep. 21, 2017, 13 pages, Oracle.com.

“Java Platform Module System”, online: https://en.wikipedia.org/wiki/Java_Platform_Module_System, Apr. 2019, 3 pages, Wikimedia Foundation, Inc.

“Java Platform, Standard Edition Tools Reference”, online: <https://docs.oracle.com/javase/9/tools/jdeps.htm#JSWOR690>, Release 9, Oct. 2017, 6 pages, Oracle.com.

* cited by examiner

100 ↙

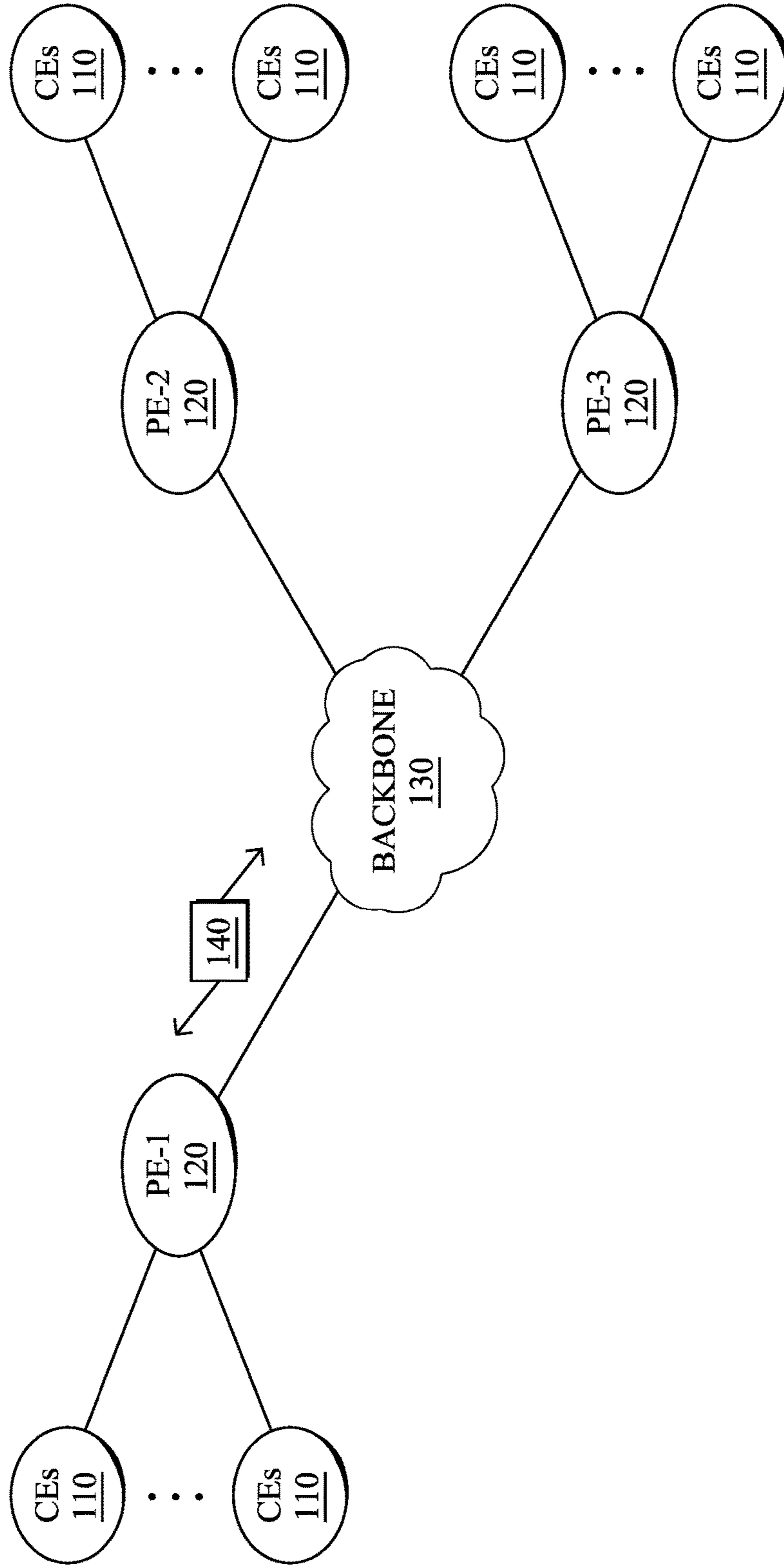


FIG. 1A

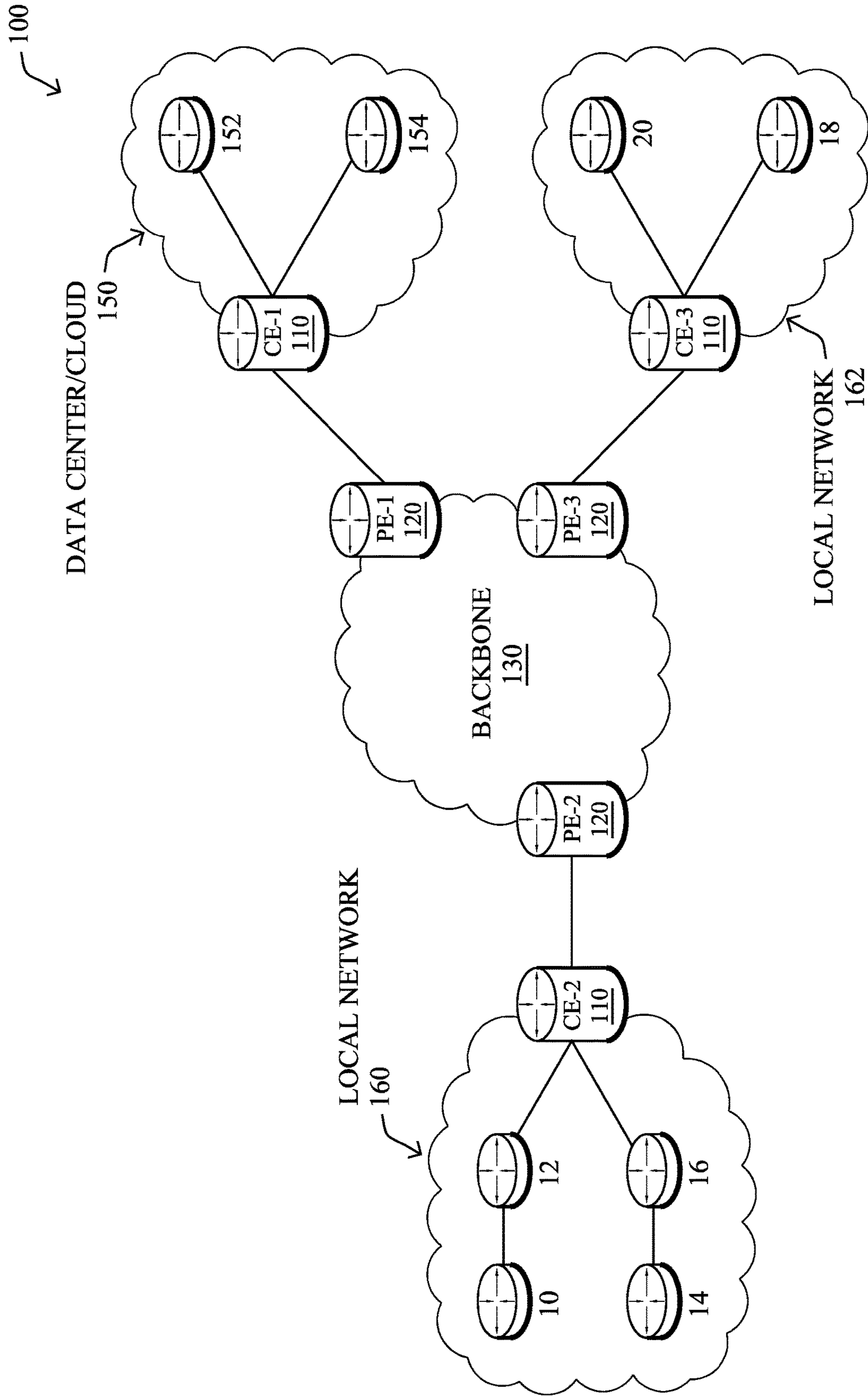


FIG. 1B

DEVICE/NODE 200

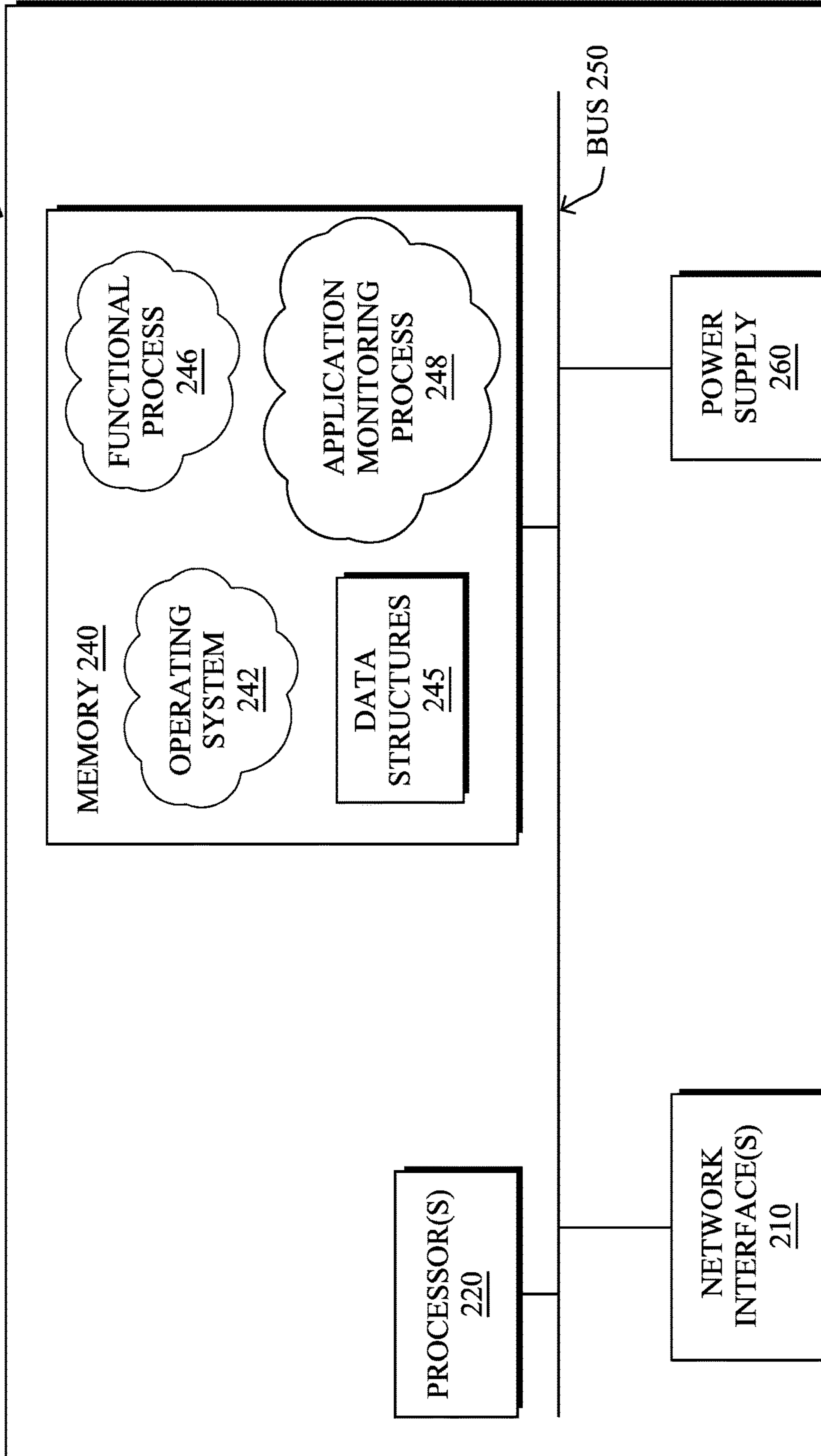


FIG. 2

300 ↘

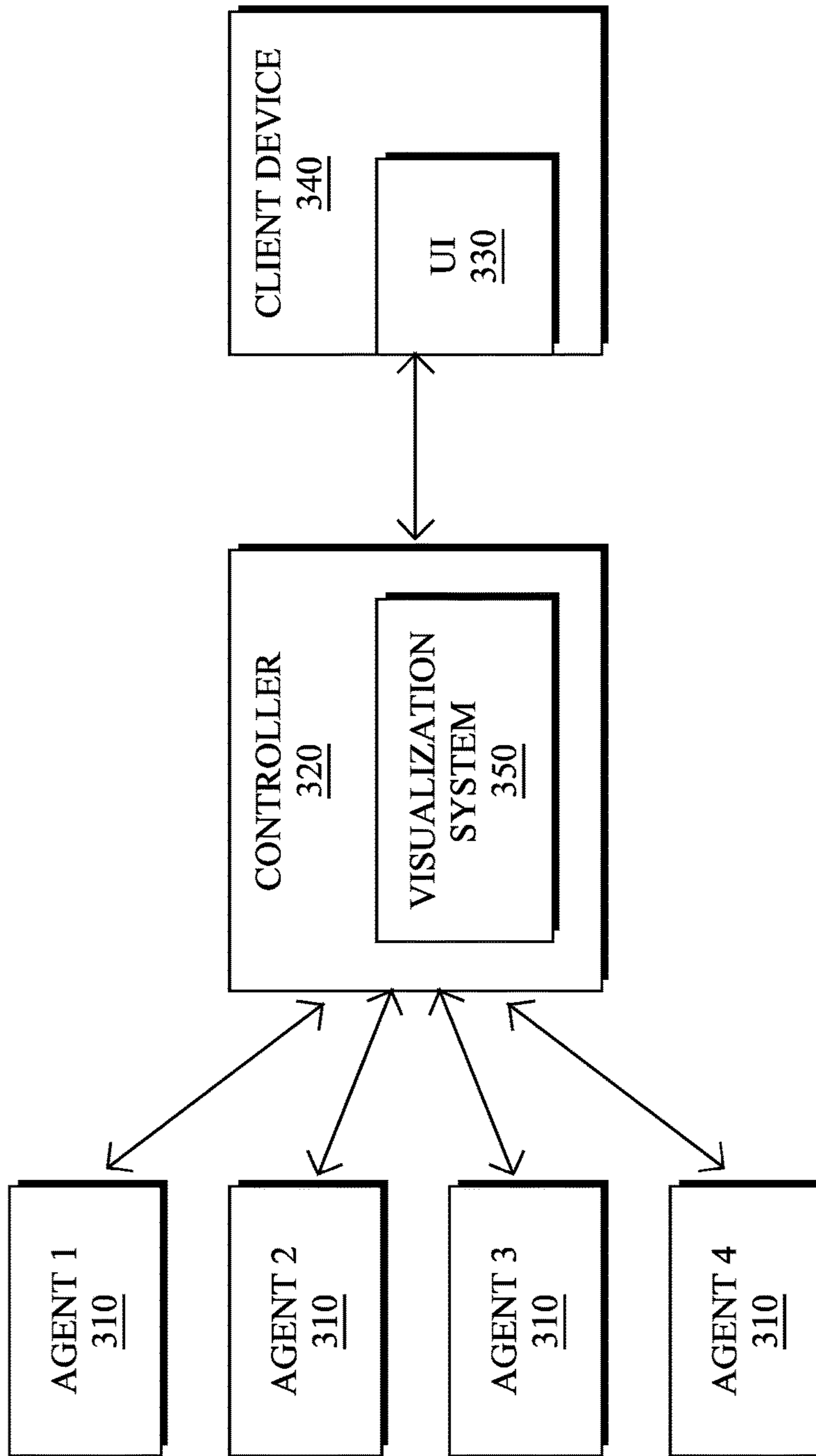


FIG. 3

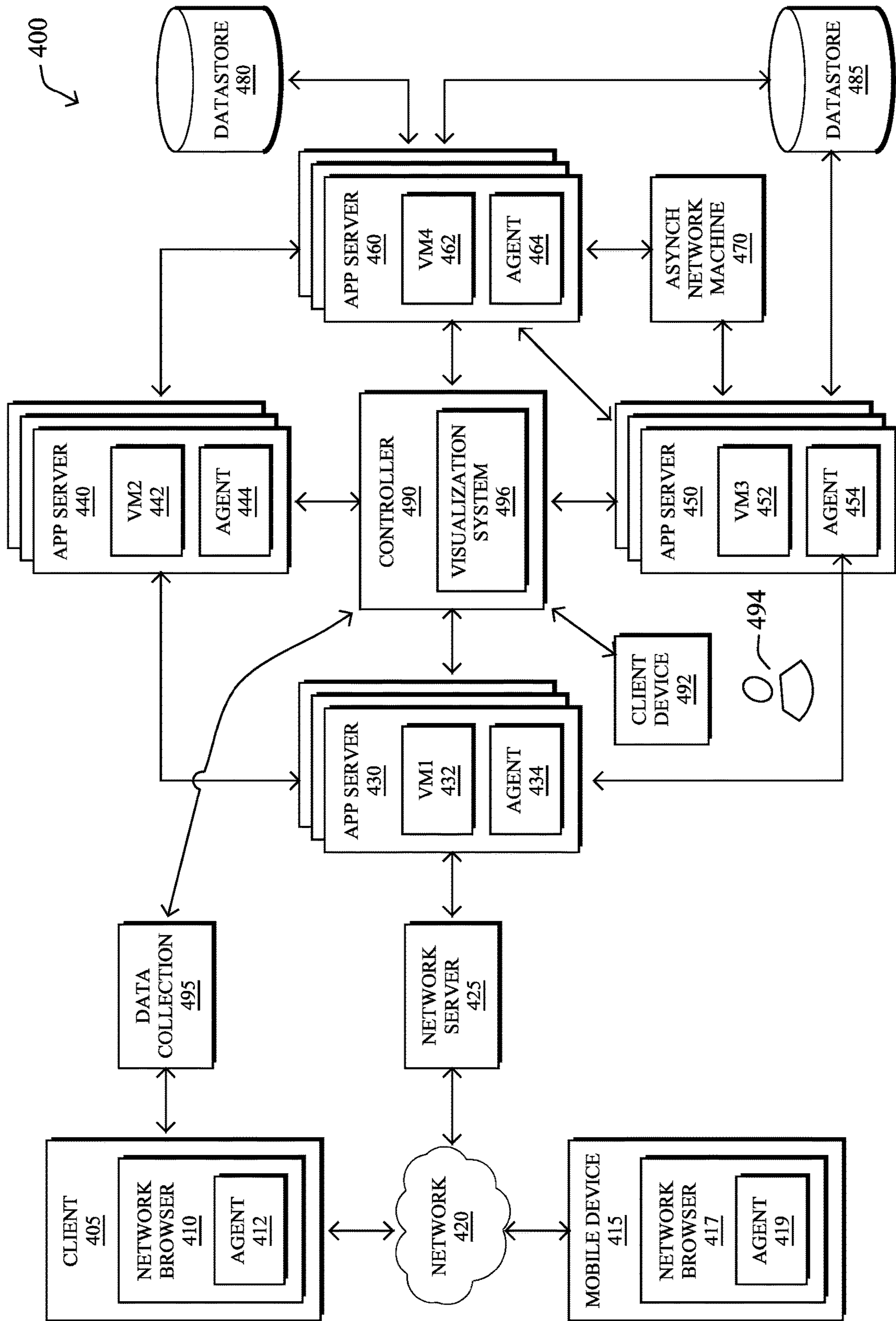


FIG. 4

500 ↙

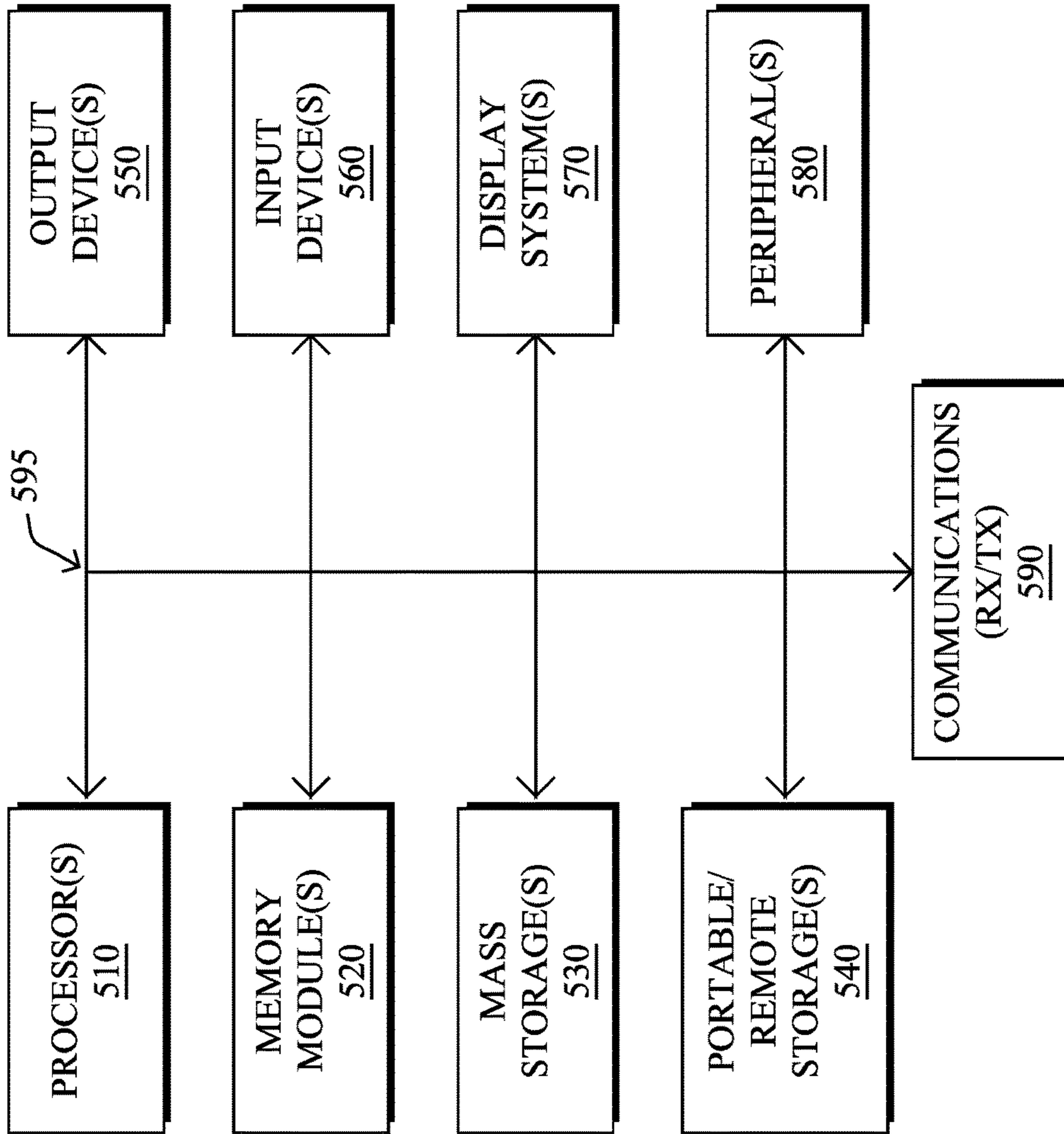


FIG. 5

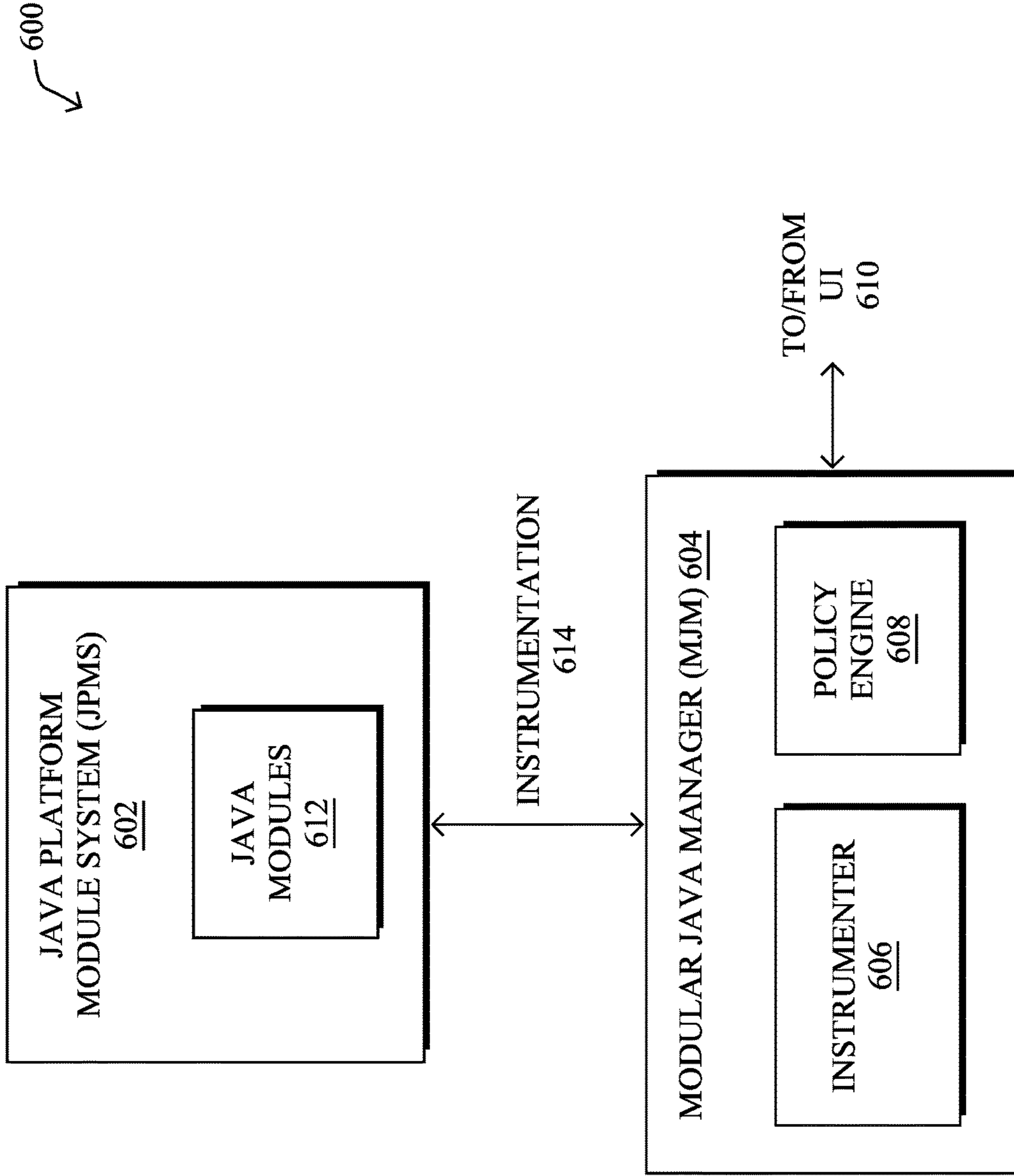


FIG. 6

700
↙

```
thulick-osx:JavaModularityManager ted.hulick$ ./runWithOut.sh
Drivers are java.util.Collections$3@2f4d3709
Drivers module name is module java.sql
SQL Module Permissions: true true false
Exception in thread "main" java.lang.reflect.InaccessibleObjectException: Unable
to make field private static volatile java.io.PrintWriter java.sql.DriverManager.
logWriter accessible: module java.sql does not "opens java.sql" to module app1
module
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible
(AccessibleObject.java:344)
    at java.base/java.lang.reflect.AccessibleObject.checkCanSetAccessible
(AccessibleObject.java:284)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:
176)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:170)
    at app1 module/app1.App1.testSQL(App1.java:59)
    at app1 module/app1.App1.main(App1.java:14)
```

FIG. 7A

710
↙

```
[thulick-osx:JavaModularityManager ted.hulick$ ./runWith.sh  
Drivers are java.util.Collections$3@74f0ea38  
Drivers module name is module java.sql  
SQL Module Permissions: true true true  
Value for logWriter is null  
■
```

FIG. 7B

800

Name	Layer	Loader	Opens	Requir
jdk.proxy2	None	JavassistJavaAgent-Class-Loader	[]	[manda java.ba
jdk.proxy1	None	null	[]	[manda java.ba
jdk.management.jfr	java.lang.ModuleLayer.aec6354	null	[]	[jdk.jfr java.m manda java.ba jdk.ma
jdk.management	java.lang.ModuleLayer.aec6354	null	[]	[manda java.ba transiti java.m
jdk.localdata	java.lang.ModuleLayer.aec6354	jdk.internal.loader.ClassLoaders\$PlatformClassLoader@cac736f	[]	[manda java.ba
jdk.jfr	java.lang.ModuleLayer.aec6354	null	[]	[manda java.ba

FIG. 8A

810

The screenshot shows a web browser interface. The address bar contains the URL `localhost:8000/modulemanager?openchecks`. Below the address bar is a navigation bar with icons for back, forward, refresh, and home, along with a star icon for bookmarks. A grid icon labeled 'Apps' is also visible. The main content area displays the title 'Cached Module Permission Check Summary for Opens (exposes all APIs via reflection to another module)' and a table with the following data:

ModuleOpening	ModuleOpenTo	PackageName	Calls	Result	PolicyOverride
java.sql	app1 module	java.sql	2	false	true

FIG. 8B

820

← →
↻
localhost:8000/modulemanager?readchecks
☆
🔍
📱
⌵

📱
Apps

Cached Module Permission Check Summary for Reads (allows to use exported APIs - equivalent to a runtime requires in module-info)

ModuleReading	ModuleCanBeRead	PackageName	Calls	Result	PolicyOverride
java.management	java.base	None	6	true	false
java.sql	java.base	None	2	true	false
app1module	java.base	None	2	true	false
jdk.httpserver	java.base	None	12	true	false
app1module	java.sql	None	1	true	false

FIG. 8C

830

← → ↻
localhost:8000/modulemanager?exportchecks
☆
🔍
📄

>>

⌵
Apps

Cached Module Permission Check Summary for Exports (exposes public APIs to another module)

ModuleExporting	ModuleExportingTo	PackageName	Calls	Result	PolicyOverride
java.base	AgentClassLoader	java.lang.module	80	true	false
java.base	jdk.httpserver	java.lang.invoke	7	true	false
java.base	java.management	java.lang.invoke	6	true	false
java.base	jdk.httpserver	java.lang	3	true	false
java.base	app1module	java.lang.invoke	2	true	false
java.base	jdk.httpserver	java.util	2	true	false
java.sql	app1module	java.sql	2	true	false
java.base	java.sql	java.lang.invoke	2	true	false
java.base	java.management	java.lang	18	true	false
java.management	null	sun.management.spi	1	false	true
java.base	null	java.lang	1	true	false

FIG. 8D

840

← →
↻
localhost:8000/modulemanager?cachedchecks
☆
↗
🏠

Apps
^

Cached Module Permission Checks

Stamp	Type	Module	OtherModule	PackageName	Result
Tue Jul 02 13:17:34 CDT 2019	isExported	java.base	jdk.httpsrver	java.lang.invoke	0
Tue Jul 02 13:17:34 CDT 2019	canRead	jdk.httpsrver	java.base	None	0
Tue Jul 02 13:17:34 CDT 2019	isExported	java.base	jdk.httpsrver	java.lang.invoke	0
Tue Jul 02 13:17:34 CDT 2019	canRead	jdk.httpsrver	java.base	None	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0
Tue Jul 02 10:39:49 CDT 2019	isExported	java.base	AgentClassLoader	java.lang.module	0

...

FIG. 8E

850 ↙


← → ↻ ⓘ localhost:8000/modulemanager?commandfixes
 Apps
The following commands can also be added to the Java Command Line to apply the policy: -- add-exports java.management/sun.management.spi=null -- add-opens java.sql/java.sql=app1 module

FIG. 8F

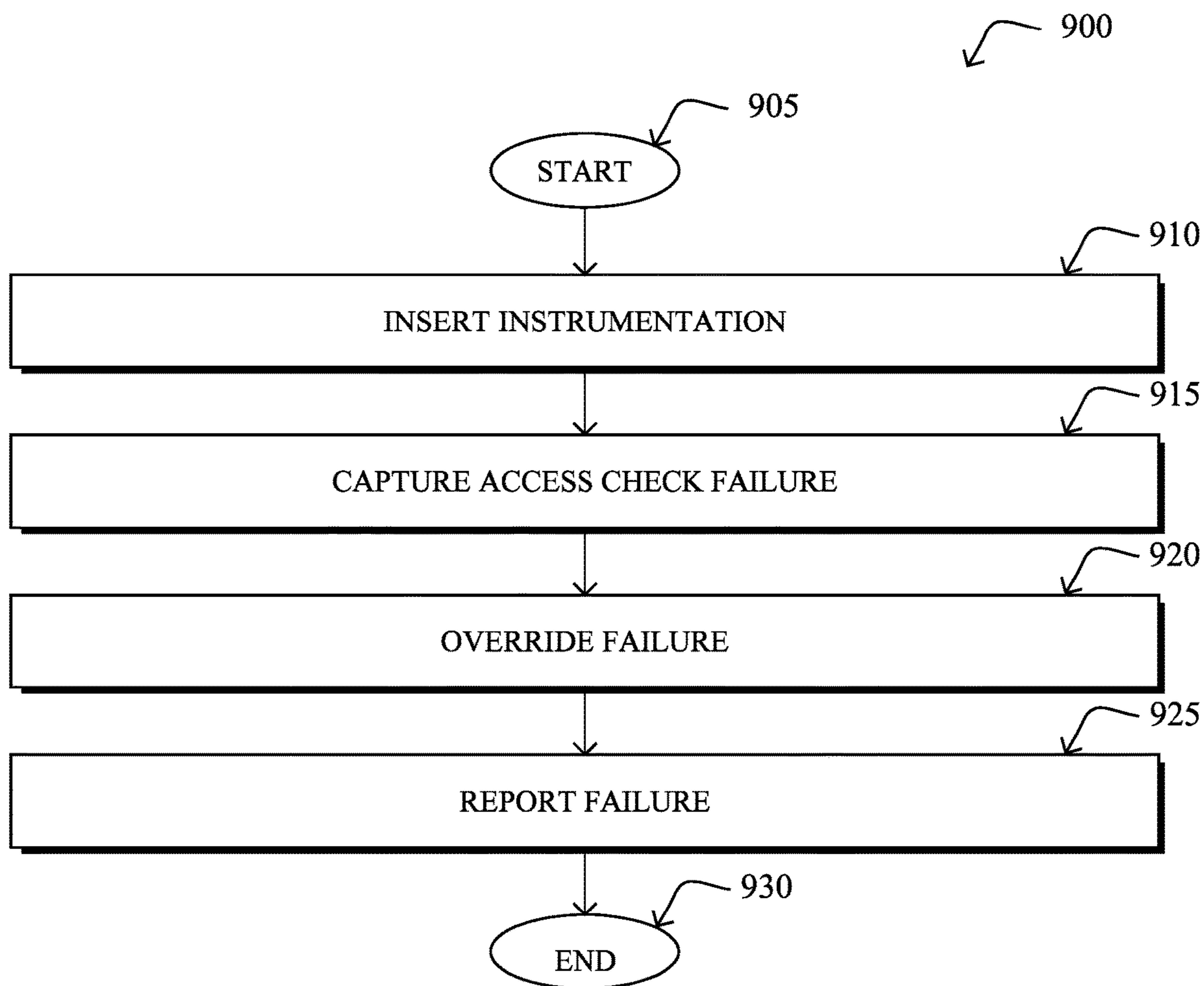


FIG. 9

1

**MODULAR JAVA MANAGER PERFORMING
CAPTURING AND OVERRIDING OF
ACCESS CHECK FAILURES FOR
INTER-MODULE OPERATIONS BETWEEN
JAVA MODULES BY AN AGENT USING
INSERTED INSTRUMENTATION**

TECHNICAL FIELD

The present disclosure relates generally to computer systems, and, more particularly, to a modular Java manager.

BACKGROUND

Many applications today rely on the Java programming language, due to its platform independence, application programming interface (API) support, multithreading capabilities, and the like. Java version 9 (Jigsaw) introduced the concept of a “module,” which is essentially a named groupings of related Java packages (e.g., code) and resources (e.g., images, XML files, etc.) that is responsible for a particular functionality/service within an application. A Java module may also share dependencies with other Java modules, which are defined as part of the module.

A key feature of Java modules is the ability to restrict access between modules. Indeed, in Java version 8 and prior, the Reflection API could be used to access all classes in a package, including its private classes, regardless of the access specifier used. With Java modules, classes in packages within a module need to have permission to access a class and to perform reflection on a class. This is done by a module “exporting” itself and certain packages to another module that “reads” that module and its exported packages. In addition, a module can “open” itself to another module, to allow reflection.

Despite the advantages of the changes introduced in Java version 9, the overwhelming majority of applications still rely on Java version 8 or prior. This is primarily due to concerns about adding even more complexity to already complex applications.

BRIEF DESCRIPTION OF THE DRAWINGS

The embodiments herein may be better understood by referring to the following description in conjunction with the accompanying drawings in which like reference numerals indicate identically or functionally similar elements, of which:

FIGS. 1A-1B illustrate an example computer network;
FIG. 2 illustrates an example computing device/node;
FIG. 3 illustrates an example application intelligence platform;

FIG. 4 illustrates an example system for implementing the example application intelligence platform;

FIG. 5 illustrates an example computing system implementing the disclosed technology;

FIG. 6 illustrates an example architecture for a modular Java manager (MJM) in accordance with one or more embodiments described herein;

FIGS. 7A-7B illustrate example screen captures of the execution of a test application with and without the use of an MJM in accordance with one or more embodiments described herein;

FIGS. 8A-8F illustrate example screen captures of the outputs of a prototype MJM system in accordance with one or more embodiments described herein; and

2

FIG. 9 illustrates an example simplified procedure for assessing Java 9+ compliance in accordance with one or more embodiments described herein.

5 DESCRIPTION OF EXAMPLE EMBODIMENTS

Overview

According to one or more embodiments of the disclosure, an agent inserts instrumentation into a Java Platform Module System in which a plurality of Java-based modules of an application is executed. The agent captures, using the instrumentation, an access check failure for an inter-module operation between the Java modules. The agent overrides, using the instrumentation, the access check failure. The agent reports the captured access check failure to a user interface.

Other embodiments are described below, and this overview is not meant to limit the scope of the present disclosure.

DESCRIPTION

A computer network is a geographically distributed collection of nodes interconnected by communication links and segments for transporting data between end nodes, such as personal computers and workstations, or other devices, such as sensors, etc. Many types of networks are available, ranging from local area networks (LANs) to wide area networks (WANs). LANs typically connect the nodes over dedicated private communications links located in the same general physical location, such as a building or campus. WANs, on the other hand, typically connect geographically dispersed nodes over long-distance communications links, such as common carrier telephone lines, optical lightpaths, synchronous optical networks (SONET), synchronous digital hierarchy (SDH) links, or Powerline Communications (PLC), and others. The Internet is an example of a WAN that connects disparate networks throughout the world, providing global communication between nodes on various networks. Other types of networks, such as field area networks (FANs), neighborhood area networks (NANs), personal area networks (PANs), enterprise networks, etc. may also make up the components of any given computer network.

The nodes typically communicate over the network by exchanging discrete frames or packets of data according to predefined protocols, such as the Transmission Control Protocol/Internet Protocol (TCP/IP). In this context, a protocol consists of a set of rules defining how the nodes interact with each other. Computer networks may be further interconnected by an intermediate network node, such as a router, to extend the effective “size” of each network.

Smart object networks, such as sensor networks, in particular, are a specific type of network having spatially distributed autonomous devices such as sensors, actuators, etc., that cooperatively monitor physical or environmental conditions at different locations, such as, e.g., energy/power consumption, resource consumption (e.g., water/gas/etc. for advanced metering infrastructure or “AMI” applications) temperature, pressure, vibration, sound, radiation, motion, pollutants, etc. Other types of smart objects include actuators, e.g., responsible for turning on/off an engine or perform any other actions. Sensor networks, a type of smart object network, are typically shared-media networks, such as wireless or power-line communication networks. That is, in addition to one or more sensors, each sensor device (node) in a sensor network may generally be equipped with a radio transceiver or other communication port, a microcontroller,

and an energy source, such as a battery. Generally, size and cost constraints on smart object nodes (e.g., sensors) result in corresponding constraints on resources such as energy, memory, computational speed and bandwidth.

FIG. 1A is a schematic block diagram of an example computer network **100** illustratively comprising nodes/devices, such as a plurality of routers/devices interconnected by links or networks, as shown. For example, customer edge (CE) routers **110** may be interconnected with provider edge (PE) routers **120** (e.g., PE-1, PE-2, and PE-3) in order to communicate across a core network, such as an illustrative network backbone **130**. For example, routers **110**, **120** may be interconnected by the public Internet, a multiprotocol label switching (MPLS) virtual private network (VPN), or the like. Data packets **140** (e.g., traffic/messages) may be exchanged among the nodes/devices of the computer network **100** over links using predefined network communication protocols such as the Transmission Control Protocol/Internet Protocol (TCP/IP), User Datagram Protocol (UDP), Asynchronous Transfer Mode (ATM) protocol, Frame Relay protocol, or any other suitable protocol. Those skilled in the art will understand that any number of nodes, devices, links, etc. may be used in the computer network, and that the view shown herein is for simplicity.

In some implementations, a router or a set of routers may be connected to a private network (e.g., dedicated leased lines, an optical network, etc.) or a virtual private network (VPN), such as an MPLS VPN thanks to a carrier network, via one or more links exhibiting very different network and service level agreement characteristics.

FIG. 1B illustrates an example of network **100** in greater detail, according to various embodiments. As shown, network backbone **130** may provide connectivity between devices located in different geographical areas and/or different types of local networks. For example, network **100** may comprise local/branch networks **160**, **162** that include devices/nodes **10-16** and devices/nodes **18-20**, respectively, as well as a data center/cloud environment **150** that includes servers **152-154**. Notably, local networks **160-162** and data center/cloud environment **150** may be located in different geographic locations. Servers **152-154** may include, in various embodiments, any number of suitable servers or other cloud-based resources. As would be appreciated, network **100** may include any number of local networks, data centers, cloud environments, devices/nodes, servers, etc.

In some embodiments, the techniques herein may be applied to other network topologies and configurations. For example, the techniques herein may be applied to peering points with high-speed links, data centers, etc. Furthermore, in various embodiments, network **100** may include one or more mesh networks, such as an Internet of Things network. Loosely, the term “Internet of Things” or “IoT” refers to uniquely identifiable objects (things) and their virtual representations in a network-based architecture. In particular, the next frontier in the evolution of the Internet is the ability to connect more than just computers and communications devices, but rather the ability to connect “objects” in general, such as lights, appliances, vehicles, heating, ventilating, and air-conditioning (HVAC), windows and window shades and blinds, doors, locks, etc. The “Internet of Things” thus generally refers to the interconnection of objects (e.g., smart objects), such as sensors and actuators, over a computer network (e.g., via IP), which may be the public Internet or a private network.

Notably, shared-media mesh networks, such as wireless networks, are often on what is referred to as Low-Power and Lossy Networks (LLNs), which are a class of network in

which both the routers and their interconnect are constrained: LLN routers typically operate with constraints, e.g., processing power, memory, and/or energy (battery), and their interconnects are characterized by, illustratively, high loss rates, low data rates, and/or instability. LLNs are comprised of anything from a few dozen to thousands or even millions of LLN routers, and support point-to-point traffic (between devices inside the LLN), point-to-multipoint traffic (from a central control point such as the root node to a subset of devices inside the LLN), and multipoint-to-point traffic (from devices inside the LLN towards a central control point). Often, an IoT network is implemented with an LLN-like architecture. For example, as shown, local network **160** may be an LLN in which CE-2 operates as a root node for nodes/devices **10-16** in the local mesh, in some embodiments.

FIG. 2 is a schematic block diagram of an example computing device (e.g., apparatus) **200** that may be used with one or more embodiments described herein, e.g., as any of the devices shown in FIGS. 1A-1B above, and particularly as specific devices as described further below. The device may comprise one or more network interfaces **210** (e.g., wired, wireless, etc.), at least one processor **220**, and a memory **240** interconnected by a system bus **250**, as well as a power supply **260** (e.g., battery, plug-in, etc.).

The network interface(s) **210** contain the mechanical, electrical, and signaling circuitry for communicating data over links coupled to the network **100**, e.g., providing a data connection between device **200** and the data network, such as the Internet. The network interfaces may be configured to transmit and/or receive data using a variety of different communication protocols. For example, interfaces **210** may include wired transceivers, wireless transceivers, cellular transceivers, or the like, each to allow device **200** to communicate information to and from a remote computing device or server over an appropriate network. The same network interfaces **210** also allow communities of multiple devices **200** to interconnect among themselves, either peer-to-peer, or up and down a hierarchy. Note, further, that the nodes may have two different types of network connections **210**, e.g., wireless and wired/physical connections, and that the view herein is merely for illustration. Also, while the network interface **210** is shown separately from power supply **260**, for devices using powerline communication (PLC) or Power over Ethernet (PoE), the network interface **210** may communicate through the power supply **260**, or may be an integral component of the power supply.

The memory **240** comprises a plurality of storage locations that are addressable by the processor **220** and the network interfaces **210** for storing software programs and data structures associated with the embodiments described herein. The processor **220** may comprise hardware elements or hardware logic adapted to execute the software programs and manipulate the data structures **245**. An operating system **242**, portions of which are typically resident in memory **240** and executed by the processor, functionally organizes the device by, among other things, invoking operations in support of software processes and/or services executing on the device. These software processes and/or services may comprise one or more functional processes **246**, and on certain devices, an illustrative “web application security communication” process **248**, as described herein. Notably, functional processes **246**, when executed by processor(s) **220**, cause each particular device **200** to perform the various functions corresponding to the particular device’s purpose and general configuration. For example, a router would be configured to operate as a router, a server would be configured to operate

as a server, an access point (or gateway) would be configured to operate as an access point (or gateway), a client device would be configured to operate as a client device, and so on.

It will be apparent to those skilled in the art that other processor and memory types, including various computer-readable media, may be used to store and execute program instructions pertaining to the techniques described herein. Also, while the description illustrates various processes, it is expressly contemplated that various processes may be embodied as modules configured to operate in accordance with the techniques herein (e.g., according to the functionality of a similar process). Further, while the processes have been shown separately, those skilled in the art will appreciate that processes may be routines or modules within other processes.

Application Intelligence Platform

The embodiments herein relate to an application intelligence platform for application performance management. In one aspect, as discussed with respect to FIGS. 3-5 below, performance within a networking environment may be monitored, specifically by monitoring applications and entities (e.g., transactions, tiers, nodes, and machines) in the networking environment using agents installed at individual machines at the entities. As an example, applications may be configured to run on one or more machines (e.g., a customer will typically run one or more nodes on a machine, where an application consists of one or more tiers, and a tier consists of one or more nodes). The agents collect data associated with the applications of interest and associated nodes and machines where the applications are being operated. Examples of the collected data may include performance data (e.g., metrics, metadata, etc.) and topology data (e.g., indicating relationship information). The agent-collected data may then be provided to one or more servers or controllers to analyze the data.

FIG. 3 is a block diagram of an example application intelligence platform 300 that can implement one or more aspects of the techniques herein. The application intelligence platform is a system that monitors and collects metrics of performance data for an application environment being monitored. At the simplest structure, the application intelligence platform includes one or more agents 310 and one or more servers/controllers 320. Note that while FIG. 3 shows four agents (e.g., Agent 1 through Agent 4) communicatively linked to a single controller, the total number of agents and controllers can vary based on a number of factors including the number of applications monitored, how distributed the application environment is, the level of monitoring desired, the level of user experience desired, and so on.

The controller 320 is the central processing and administration server for the application intelligence platform. The controller 320 serves a browser-based user interface (UI) 330 that is the primary interface for monitoring, analyzing, and troubleshooting the monitored environment. The controller 320 can control and manage monitoring of business transactions (described below) distributed over application servers. Specifically, the controller 320 can receive runtime data from agents 310 (and/or other coordinator devices), associate portions of business transaction data, communicate with agents to configure collection of runtime data, and provide performance data and reporting through the interface 330. The interface 330 may be viewed as a web-based interface viewable by a client device 340. In some imple-

mentations, a client device 340 can directly communicate with controller 320 to view an interface for monitoring data. The controller 320 can include a visualization system 350 for displaying the reports and dashboards related to the disclosed technology. In some implementations, the visualization system 350 can be implemented in a separate machine (e.g., a server) different from the one hosting the controller 320.

Notably, in an illustrative Software as a Service (SaaS) implementation, a controller instance 320 may be hosted remotely by a provider of the application intelligence platform 300. In an illustrative on-premises (On-Prem) implementation, a controller instance 320 may be installed locally and self-administered.

The controllers 320 receive data from different agents 310 (e.g., Agents 1-4) deployed to monitor applications, databases and database servers, servers, and end user clients for the monitored environment. Any of the agents 310 can be implemented as different types of agents with specific monitoring duties. For example, application agents may be installed on each server that hosts applications to be monitored. Instrumenting an agent adds an application agent into the runtime process of the application.

Database agents, for example, may be software (e.g., a Java program) installed on a machine that has network access to the monitored databases and the controller. Database agents query the monitored databases in order to collect metrics and pass those metrics along for display in a metric browser (e.g., for database monitoring and analysis within databases pages of the controller's UI 330). Multiple database agents can report to the same controller. Additional database agents can be implemented as backup database agents to take over for the primary database agents during a failure or planned machine downtime. The additional database agents can run on the same machine as the primary agents or on different machines. A database agent can be deployed in each distinct network of the monitored environment. Multiple database agents can run under different user accounts on the same machine.

Standalone machine agents, on the other hand, may be standalone programs (e.g., standalone Java programs) that collect hardware-related performance statistics from the servers (or other suitable devices) in the monitored environment. The standalone machine agents can be deployed on machines that host application servers, database servers, messaging servers, Web servers, etc. A standalone machine agent has an extensible architecture (e.g., designed to accommodate changes).

End user monitoring (EUM) may be performed using browser agents and mobile agents to provide performance information from the point of view of the client, such as a web browser or a mobile native application. Through EUM, web use, mobile use, or combinations thereof (e.g., by real users or synthetic agents) can be monitored based on the monitoring needs. Notably, browser agents (e.g., agents 310) can include Reporters that report monitored data to the controller.

Monitoring through browser agents and mobile agents are generally unlike monitoring through application agents, database agents, and standalone machine agents that are on the server. In particular, browser agents may generally be embodied as small files using web-based technologies, such as JavaScript agents injected into each instrumented web page (e.g., as close to the top as possible) as the web page is served, and are configured to collect data. Once the web page has completed loading, the collected data may be bundled into a beacon and sent to an EUM process/cloud for

processing and made ready for retrieval by the controller. Browser real user monitoring (Browser RUM) provides insights into the performance of a web application from the point of view of a real or synthetic end user. For example, Browser RUM can determine how specific Ajax or iframe

calls are slowing down page load time and how server performance impact end user experience in aggregate or in individual cases. A mobile agent, on the other hand, may be a small piece of highly performant code that gets added to the source of the mobile application. Mobile RUM provides information on the native mobile application (e.g., iOS or Android applications) as the end users actually use the mobile application. Mobile RUM provides visibility into the functioning of the mobile application itself and the mobile application's interaction with the network used and any server-side applications with which the mobile application communicates.

Application Intelligence Monitoring: The disclosed technology can provide application intelligence data by monitoring an application environment that includes various services such as web applications served from an application server (e.g., Java virtual machine (JVM), Internet Information Services (IIS), Hypertext Preprocessor (PHP) Web server, etc.), databases or other data stores, and remote services such as message queues and caches. The services in the application environment can interact in various ways to provide a set of cohesive user interactions with the application, such as a set of user services applicable to end user customers.

Application Intelligence Modeling: Entities in the application environment (such as the JBoss service, MQSeries modules, and databases) and the services provided by the entities (such as a login transaction, service or product search, or purchase transaction) may be mapped to an application intelligence model. In the application intelligence model, a business transaction represents a particular service provided by the monitored environment. For example, in an e-commerce application, particular real-world services can include a user logging in, searching for items, or adding items to the cart. In a content portal, particular real-world services can include user requests for content such as sports, business, or entertainment news. In a stock trading application, particular real-world services can include operations such as receiving a stock quote, buying, or selling stocks.

Business Transactions: A business transaction representation of the particular service provided by the monitored environment provides a view on performance data in the context of the various tiers that participate in processing a particular request. A business transaction, which may each be identified by a unique business transaction identification (ID), represents the end-to-end processing path used to fulfill a service request in the monitored environment (e.g., adding items to a shopping cart, storing information in a database, purchasing an item online, etc.). Thus, a business transaction is a type of user-initiated action in the monitored environment defined by an entry point and a processing path across application servers, databases, and potentially many other infrastructure components. Each instance of a business transaction is an execution of that transaction in response to a particular user request (e.g., a socket call, illustratively associated with the TCP layer). A business transaction can be created by detecting incoming requests at an entry point and tracking the activity associated with request at the originating tier and across distributed components in the application environment (e.g., associating the business transaction with

a 4-tuple of a source IP address, source port, destination IP address, and destination port). A flow map can be generated for a business transaction that shows the touch points for the business transaction in the application environment. In one embodiment, a specific tag may be added to packets by application specific agents for identifying business transactions (e.g., a custom header field attached to a hypertext transfer protocol (HTTP) payload by an application agent, or by a network agent when an application makes a remote socket call), such that packets can be examined by network agents to identify the business transaction identifier (ID) (e.g., a Globally Unique Identifier (GUID) or Universally Unique Identifier (UUID)).

Performance monitoring can be oriented by business transaction to focus on the performance of the services in the application environment from the perspective of end users. Performance monitoring based on business transactions can provide information on whether a service is available (e.g., users can log in, check out, or view their data), response times for users, and the cause of problems when the problems occur.

A business application is the top-level container in the application intelligence model. A business application contains a set of related services and business transactions. In some implementations, a single business application may be needed to model the environment. In some implementations, the application intelligence model of the application environment can be divided into several business applications. Business applications can be organized differently based on the specifics of the application environment. One consideration is to organize the business applications in a way that reflects work teams in a particular organization, since role-based access controls in the Controller UI are oriented by business application.

A node in the application intelligence model corresponds to a monitored server or JVM in the application environment. A node is the smallest unit of the modeled environment. In general, a node corresponds to an individual application server, JVM, or Common Language Runtime (CLR) on which a monitoring Agent is installed. Each node identifies itself in the application intelligence model. The Agent installed at the node is configured to specify the name of the node, tier, and business application under which the Agent reports data to the Controller.

Business applications contain tiers, the unit in the application intelligence model that includes one or more nodes. Each node represents an instrumented service (such as a web application). While a node can be a distinct application in the application environment, in the application intelligence model, a node is a member of a tier, which, along with possibly many other tiers, make up the overall logical business application.

Tiers can be organized in the application intelligence model depending on a mental model of the monitored application environment. For example, identical nodes can be grouped into a single tier (such as a cluster of redundant servers). In some implementations, any set of nodes, identical or not, can be grouped for the purpose of treating certain performance metrics as a unit into a single tier.

The traffic in a business application flows among tiers and can be visualized in a flow map using lines among tiers. In addition, the lines indicating the traffic flows among tiers can be annotated with performance metrics. In the application intelligence model, there may not be any interaction among nodes within a single tier. Also, in some implementations, an application agent node cannot belong to more than one tier.

Similarly, a machine agent cannot belong to more than one tier. However, more than one machine agent can be installed on a machine.

A backend is a component that participates in the processing of a business transaction instance. A backend is not instrumented by an agent. A backend may be a web server, database, message queue, or other type of service. The agent recognizes calls to these backend services from instrumented code (called exit calls). When a service is not instrumented and cannot continue the transaction context of the call, the agent determines that the service is a backend component. The agent picks up the transaction context at the response at the backend and continues to follow the context of the transaction from there.

Performance information is available for the backend call. For detailed transaction analysis for the leg of a transaction processed by the backend, the database, web service, or other application need to be instrumented.

The application intelligence platform uses both self-learned baselines and configurable thresholds to help identify application issues. A complex distributed application has a large number of performance metrics and each metric is important in one or more contexts. In such environments, it is difficult to determine the values or ranges that are normal for a particular metric; set meaningful thresholds on which to base and receive relevant alerts; and determine what is a “normal” metric when the application or infrastructure undergoes change. For these reasons, the disclosed application intelligence platform can perform anomaly detection based on dynamic baselines or thresholds.

The disclosed application intelligence platform automatically calculates dynamic baselines for the monitored metrics, defining what is “normal” for each metric based on actual usage. The application intelligence platform uses these baselines to identify subsequent metrics whose values fall out of this normal range. Static thresholds that are tedious to set up and, in rapidly changing application environments, error-prone, are no longer needed.

The disclosed application intelligence platform can use configurable thresholds to maintain service level agreements (SLAs) and ensure optimum performance levels for system by detecting slow, very slow, and stalled transactions. Configurable thresholds provide a flexible way to associate the right business context with a slow request to isolate the root cause.

In addition, health rules can be set up with conditions that use the dynamically generated baselines to trigger alerts or initiate other types of remedial actions when performance problems are occurring or may be about to occur.

For example, dynamic baselines can be used to automatically establish what is considered normal behavior for a particular application. Policies and health rules can be used against baselines or other health indicators for a particular application to detect and troubleshoot problems before users are affected. Health rules can be used to define metric conditions to monitor, such as when the “average response time is four times slower than the baseline”. The health rules can be created and modified based on the monitored application environment.

Examples of health rules for testing business transaction performance can include business transaction response time and business transaction error rate. For example, health rule that tests whether the business transaction response time is much higher than normal can define a critical condition as the combination of an average response time greater than the default baseline by 3 standard deviations and a load greater than 50 calls per minute. In some implementations, this

health rule can define a warning condition as the combination of an average response time greater than the default baseline by 2 standard deviations and a load greater than 100 calls per minute. In some implementations, the health rule that tests whether the business transaction error rate is much higher than normal can define a critical condition as the combination of an error rate greater than the default baseline by 3 standard deviations and an error rate greater than 10 errors per minute and a load greater than 50 calls per minute. In some implementations, this health rule can define a warning condition as the combination of an error rate greater than the default baseline by 2 standard deviations and an error rate greater than 5 errors per minute and a load greater than 50 calls per minute. These are non-exhaustive and non-limiting examples of health rules and other health rules can be defined as desired by the user.

Policies can be configured to trigger actions when a health rule is violated or when any event occurs. Triggered actions can include notifications, diagnostic actions, auto-scaling capacity, running remediation scripts.

Most of the metrics relate to the overall performance of the application or business transaction (e.g., load, average response time, error rate, etc.) or of the application server infrastructure (e.g., percentage CPU busy, percentage of memory used, etc.). The Metric Browser in the controller UI can be used to view all of the metrics that the agents report to the controller.

In addition, special metrics called information points can be created to report on how a given business (as opposed to a given application) is performing. For example, the performance of the total revenue for a certain product or set of products can be monitored. Also, information points can be used to report on how a given code is performing, for example how many times a specific method is called and how long it is taking to execute. Moreover, extensions that use the machine agent can be created to report user defined custom metrics. These custom metrics are base-lined and reported in the controller, just like the built-in metrics.

All metrics can be accessed programmatically using a Representational State Transfer (REST) API that returns either the JavaScript Object Notation (JSON) or the eXtensible Markup Language (XML) format. Also, the REST API can be used to query and manipulate the application environment.

Snapshots provide a detailed picture of a given application at a certain point in time. Snapshots usually include call graphs that allow that enables drilling down to the line of code that may be causing performance problems. The most common snapshots are transaction snapshots.

FIG. 4 illustrates an example application intelligence platform (system) 400 for performing one or more aspects of the techniques herein. The system 400 in FIG. 4 includes client device 405 and 492, mobile device 415, network 420, network server 425, application servers 430, 440, 450, and 460, asynchronous network machine 470, data stores 480 and 485, controller 490, and data collection server 495. The controller 490 can include visualization system 496 for providing displaying of the report generated for performing the field name recommendations for field extraction as disclosed in the present disclosure. In some implementations, the visualization system 496 can be implemented in a separate machine (e.g., a server) different from the one hosting the controller 490.

Client device 405 may include network browser 410 and be implemented as a computing device, such as for example a laptop, desktop, workstation, or some other computing device. Network browser 410 may be a client application for

viewing content provided by an application server, such as application server **430** via network server **425** over network **420**.

Network browser **410** may include agent **412**. Agent **412** may be installed on network browser **410** and/or client **405** 5 as a network browser add-on, downloading the application to the server, or in some other manner. Agent **412** may be executed to monitor network browser **410**, the operating system of client **405**, and any other application, API, or another component of client **405**. Agent **412** may determine 10 network browser navigation timing metrics, access browser cookies, monitor code, and transmit data to data collection **495**, controller **490**, or another device. Agent **412** may perform other operations related to monitoring a request or a network at client **405** as discussed herein including report 15 generating.

Mobile device **415** is connected to network **420** and may be implemented as a portable device suitable for sending and receiving content over a network, such as for example a 20 mobile phone, smart phone, tablet computer, or other portable device. Both client device **405** and mobile device **415** may include hardware and/or software configured to access a web service provided by network server **425**.

Mobile device **415** may include network browser **417** and an agent **419**. Mobile device may also include client applica- 25 tions and other code that may be monitored by agent **419**. Agent **419** may reside in and/or communicate with network browser **417**, as well as communicate with other applications, an operating system, APIs and other hardware and software on mobile device **415**. Agent **419** may have similar 30 functionality as that described herein for agent **412** on client **405**, and may report data to data collection server **495** and/or controller **490**.

Network **420** may facilitate communication of data among different servers, devices and machines of system 35 **400** (some connections shown with lines to network **420**, some not shown). The network may be implemented as a private network, public network, intranet, the Internet, a cellular network, Wi-Fi network, VoIP network, or a combination of one or more of these networks. The network **420** 40 may include one or more machines such as load balance machines and other machines.

Network server **425** is connected to network **420** and may receive and process requests received over network **420**. Network server **425** may be implemented as one or more 45 servers implementing a network service, and may be implemented on the same machine as application server **430** or one or more separate machines. When network **420** the Internet, network server **425** may be implemented as a web server.

Application server **430** communicates with network server **425**, application servers **440** and **450**, and controller **490**. Application server **450** may also communicate with other machines and devices (not illustrated in FIG. 4). Application server **430** may host an application or portions 55 of a distributed application. The host application **432** may be in one of many platforms, such as including a Java, PHP, .Net, and Node.JS, be implemented as a Java virtual machine, or include some other host type. Application server **430** may also include one or more agents **434** (i.e., “mod- 60 ules”), including a language agent, machine agent, and network agent, and other software modules. Application server **430** may be implemented as one server or multiple servers as illustrated in FIG. 4.

Application **432** and other software on application server 65 **430** may be instrumented using byte code insertion, or byte code instrumentation (BCI), to modify the object code of the

application or other software. The instrumented object code may include code used to detect calls received by applica- tion **432**, calls sent by application **432**, and communicate with agent **434** during execution of the application. BCI may 5 also be used to monitor one or more sockets of the application and/or application server in order to monitor the socket and capture packets coming over the socket.

In some embodiments, server **430** may include applica- tions and/or code other than a virtual machine. For example, servers **430**, **440**, **450**, and **460** may each include Java code, .Net code, PHP code, Ruby code, C code, C++ or other 10 binary code to implement applications and process requests received from a remote source. References to a virtual machine with respect to an application server are intended to be for exemplary purposes only.

Agents **434** on application server **430** may be installed, downloaded, embedded, or otherwise provided on applica- tion server **430**. For example, agents **434** may be provided in server **430** by instrumentation of object code, download- 20 ing the agents to the server, or in some other manner. Agent **434** may be executed to monitor application server **430**, monitor code running in a virtual machine **432** (or other program language, such as a PHP, .Net, or C program), machine resources, network layer data, and communicate 25 with byte instrumented code on application server **430** and one or more applications on application server **430**.

Each of agents **434**, **444**, **454**, and **464** may include one or more agents, such as language agents, machine agents, and network agents. A language agent may be a type of agent 30 that is suitable to run on a particular host. Examples of language agents include a Java agent, .Net agent, PHP agent, and other agents. The machine agent may collect data from a particular machine on which it is installed. A network agent may capture network information, such as data col- 35 lected from a socket.

Agent **434** may detect operations such as receiving calls and sending requests by application server **430**, resource usage, and incoming packets. Agent **434** may receive data, process the data, for example by aggregating data into 40 metrics, and transmit the data and/or metrics to controller **490**. Agent **434** may perform other operations related to monitoring applications and application server **430** as discussed herein. For example, agent **434** may identify other applications, share business transaction data, aggregate 45 detected runtime data, and other operations.

An agent may operate to monitor a node, tier of nodes, or other entity. A node may be a software program or a hardware component (e.g., memory, processor, and so on). A tier of nodes may include a plurality of nodes which may 50 process a similar business transaction, may be located on the same server, may be associated with each other in some other way, or may not be associated with each other.

A language agent may be an agent suitable to instrument or modify, collect data from, and reside on a host. The host 55 may be a Java, PHP, .Net, Node.JS, or other type of platform. Language agents may collect flow data as well as data associated with the execution of a particular application. The language agent may instrument the lowest level of the application to gather the flow data. The flow data may 60 indicate which tier is communicating with which tier and on which port. In some instances, the flow data collected from the language agent includes a source IP, a source port, a destination IP, and a destination port. The language agent may report the application data and call chain data to a controller. The language agent may report the collected flow 65 data associated with a particular application to a network agent.

A network agent may be a standalone agent that resides on the host and collects network flow group data. The network flow group data may include a source IP, destination port, destination IP, and protocol information for network flow received by an application on which network agent is installed. The network agent may collect data by intercepting and performing packet capture on packets coming in from one or more network interfaces (e.g., so that data generated/received by all the applications using sockets can be intercepted). The network agent may receive flow data from a language agent that is associated with applications to be monitored. For flows in the flow group data that match flow data provided by the language agent, the network agent rolls up the flow data to determine metrics such as TCP throughput, TCP loss, latency, and bandwidth. The network agent may then report the metrics, flow group data, and call chain data to a controller. The network agent may also make system calls at an application server to determine system information, such as for example a host status check, a network status check, socket status, and other information.

A machine agent, which may be referred to as an infrastructure agent, may reside on the host and collect information regarding the machine which implements the host. A machine agent may collect and generate metrics from information such as processor usage, memory usage, and other hardware information.

Each of the language agent, network agent, and machine agent may report data to the controller. Controller 490 may be implemented as a remote server that communicates with agents located on one or more servers or machines. The controller may receive metrics, call chain data and other data, correlate the received data as part of a distributed transaction, and report the correlated data in the context of a distributed application implemented by one or more monitored applications and occurring over one or more monitored networks. The controller may provide reports, one or more user interfaces, and other information for a user.

Agent 434 may create a request identifier for a request received by server 430 (for example, a request received by a client 405 or 415 associated with a user or another source). The request identifier may be sent to client 405 or mobile device 415, whichever device sent the request. In embodiments, the request identifier may be created when data is collected and analyzed for a particular business transaction.

Each of application servers 440, 450, and 460 may include an application and agents. Each application may run on the corresponding application server. Each of applications 442, 452, and 462 on application servers 440-460 may operate similarly to application 432 and perform at least a portion of a distributed business transaction. Agents 444, 454, and 464 may monitor applications 442-462, collect and process data at runtime, and communicate with controller 490. The applications 432, 442, 452, and 462 may communicate with each other as part of performing a distributed transaction. Each application may call any application or method of another virtual machine.

Asynchronous network machine 470 may engage in asynchronous communications with one or more application servers, such as application server 450 and 460. For example, application server 450 may transmit several calls or messages to an asynchronous network machine. Rather than communicate back to application server 450, the asynchronous network machine may process the messages and eventually provide a response, such as a processed message, to application server 460. Because there is no return message

from the asynchronous network machine to application server 450, the communications among them are asynchronous.

Data stores 480 and 485 may each be accessed by application servers such as application server 460. Data store 485 may also be accessed by application server 450. Each of data stores 480 and 485 may store data, process data, and return queries received from an application server. Each of data stores 480 and 485 may or may not include an agent.

Controller 490 may control and manage monitoring of business transactions distributed over application servers 430-460. In some embodiments, controller 490 may receive application data, including data associated with monitoring client requests at client 405 and mobile device 415, from data collection server 495. In some embodiments, controller 490 may receive application monitoring data and network data from each of agents 412, 419, 434, 444, and 454 (also referred to herein as “application monitoring agents”). Controller 490 may associate portions of business transaction data, communicate with agents to configure collection of data, and provide performance data and reporting through an interface. The interface may be viewed as a web-based interface viewable by client device 492, which may be a mobile device, client device, or any other platform for viewing an interface provided by controller 490. In some embodiments, a client device 492 may directly communicate with controller 490 to view an interface for monitoring data.

Client device 492 may include any computing device, including a mobile device or a client computer such as a desktop, work station or other computing device. Client computer 492 may communicate with controller 490 to create and view a custom interface. In some embodiments, controller 490 provides an interface for creating and viewing the custom interface as a content page, e.g., a web page, which may be provided to and rendered through a network browser application on client device 492.

Applications 432, 442, 452, and 462 may be any of several types of applications. Examples of applications that may implement applications 432-462 include a Java, PHP, .Net, Node.JS, and other applications.

FIG. 5 is a block diagram of a computer system 500 for implementing the present technology, which is a specific implementation of device 200 of FIG. 2 above. System 500 of FIG. 5 may be implemented in the contexts of the likes of clients 405, 492, network server 425, servers 430, 440, 450, 460, asynchronous network machine 470, and controller 490 of FIG. 4. (Note that the specifically configured system 500 of FIG. 5 and the customized device 200 of FIG. 2 are not meant to be mutually exclusive, and the techniques herein may be performed by any suitably configured computing device.)

The computing system 500 of FIG. 5 includes one or more processors 510 and memory 520. Main memory 520 stores, in part, instructions and data for execution by processor 510. Main memory 520 can store the executable code when in operation. The system 500 of FIG. 5 further includes a mass storage device 530, portable storage medium drive(s) 540, output devices 550, user input devices 560, a graphics display 570, and peripheral devices 580.

The components shown in FIG. 5 are depicted as being connected via a single bus 590. However, the components may be connected through one or more data transport means. For example, processor unit 510 and main memory 520 may be connected via a local microprocessor bus, and the mass storage device 530, peripheral device(s) 580, portable or remote storage device 540, and display system 570 may be connected via one or more input/output (I/O) buses.

Mass storage device **530**, which may be implemented with a magnetic disk drive or an optical disk drive, is a non-volatile storage device for storing data and instructions for use by processor unit **510**. Mass storage device **530** can store the system software for implementing embodiments of the present disclosure for purposes of loading that software into main memory **520**.

Portable storage device **540** operates in conjunction with a portable non-volatile storage medium, such as a compact disk, digital video disk, magnetic disk, flash storage, etc. to input and output data and code to and from the computer system **500** of FIG. **5**. The system software for implementing embodiments of the present disclosure may be stored on such a portable medium and input to the computer system **500** via the portable storage device **540**.

Input devices **560** provide a portion of a user interface. Input devices **560** may include an alpha-numeric keypad, such as a keyboard, for inputting alpha-numeric and other information, or a pointing device, such as a mouse, a trackball, stylus, or cursor direction keys. Additionally, the system **500** as shown in FIG. **5** includes output devices **550**. Examples of suitable output devices include speakers, printers, network interfaces, and monitors.

Display system **570** may include a liquid crystal display (LCD) or other suitable display device. Display system **570** receives textual and graphical information, and processes the information for output to the display device.

Peripherals **580** may include any type of computer support device to add additional functionality to the computer system. For example, peripheral device(s) **580** may include a modem or a router.

The components contained in the computer system **500** of FIG. **5** can include a personal computer, hand held computing device, telephone, mobile computing device, workstation, server, minicomputer, mainframe computer, or any other computing device. The computer can also include different bus configurations, networked platforms, multiprocessor platforms, etc. Various operating systems can be used including Unix, Linux, Windows, Apple OS, and other suitable operating systems, including mobile versions.

When implementing a mobile device such as smart phone or tablet computer, the computer system **500** of FIG. **5** may include one or more antennas, radios, and other circuitry for communicating over wireless signals, such as for example communication using Wi-Fi, cellular, or other wireless signals.

As noted above, many applications today rely on the Java programming language, due to its platform independence, application programming interface (API) support, multithreading capabilities, and the like. Java version 9 (Jigsaw) introduced the concept of a “module,” which is essentially a named groupings of related Java packages (e.g., code) and resources (e.g., images, XML files, etc.) that is responsible for a particular functionality/service within an application. A Java module may also share dependencies with other Java modules, which are defined as part of the module.

A key feature of Java modules is the ability to restrict access between modules. Indeed, in Java version 8 and prior, the Reflection API could be used to access all classes in a package, including its private classes, regardless of the access specifier used. With Java modules, classes in packages within a module need to have permission to access a class and to perform reflection on a class. This is done by a module “exporting” itself and certain packages to another module that “reads” that module and its exported packages. In addition, a module can “open” itself to another module, to allow reflection.

Despite the advantages of the changes introduced in Java version 9, the overwhelming majority of applications still rely on Java version 8 or prior. This is primarily due to concerns about adding even more complexity to already complex applications.

Modular Java Manager

The techniques herein introduce a Modular Java Manager (MJM) that is designed to help application developers and stakeholders migrate a Java application to Java version 9 or greater (collectively referred to herein as “Java 9+”), in a simplified manner. In some aspects, the MJM may operate as a Java agent that provides an audit trail of changes that need to be made to a Java application to become compliant with Java 9+. The MJM is able to do so in a manner that does not require the application to fail to obtain the full list of changes. This is in contrast to approaches that identify changes, one failure at a time. In further aspects, the MJM also provides the ability to “script” an easy to use policy, or build it automatically, as a substitute for the Oracle module-info.java files and/or the command line arguments. This can be setup to monitor the module activities in production and prevent production failures due to module permission rejections (i.e., an access check failure), while also generating notifications of any problems that arise.

FIG. **6** illustrates an example architecture **600** for a modular Java manager (MJM), in accordance with one or more embodiments described herein. At the core of architecture **600** is MJM **604** that interacts with a Java Platform Module System **602**, to implement the techniques herein. In addition, MJM **604** may communicate with a user interface (UI) **610**, such as a display or the like, to convey information to a user and receive parameters and other control commands therefrom.

As would be appreciated, Java Platform Module System (JPMS) **602** is a platform introduced in Java 9 and used in all subsequent versions to date, to support the use of Java modules within an application, such as Java modules **612**. In general, a Java module **612** may include the following information as part of a module descriptor:

- A name that uniquely identifies the module.
- A set of dependencies between that module and those on which it depends.
- A listing of the packages that it makes available to other modules via export. Note that this must be done explicitly and that a package is implicitly unavailable to other modules, by default.
- The services that are offered by the module.
- The services that the module consumes.
- The other modules that are allowed to use reflection with the module.

In addition to the module descriptor, each Java module **612** may include any number of related packages (e.g., code) and, potentially, other resources (e.g., images, XML, etc.), as well.

More specifically, a module descriptor for a Java module **612** may utilize any or all of the following directives:

- exports—this directive specifies the packages of the module that are accessible by other modules.
- uses—this directive specifies which service(s) are used by the module. In general, a service is an object for a class that implements an interface or extends the abstract class specified in this directive.

provides—this directive specifies that a module provides a particular service (e.g., the interface or abstract class from the uses directive), as well as the service provider class that implements it.

opens—this directive specifies the package(s) of the module that are accessible to other modules. Depending on its use, this directive can be used to allow all packages in the module to be accessed during runtime or used to limit runtime access by specified modules to certain modules.

According to various embodiments, MJM 604 may include any or all of the following components: an instrumenter 606 responsible for inserting instrumentation 614 into JPMS 602 and a policy engine 608 responsible for the implementation of a module policy. As would be appreciated, the functionalities of components 606-608 may be combined, omitted, or implemented in a distributed manner, as desired.

According to various embodiments, MJM 604 may also include a policy engine 608 that allows a user of UI 610 to define or adjust a module policy. In turn, MJM 604 may compare the information logged by instrumentation 614 to the module policy, to determine what action, if any should be taken. More specifically, the policy may control whether a given decision within JPMS 602 should be overridden and, if so, allow the operation. For example, if any of the methods in Table 1 return a value of ‘false,’ the instrumentation 614 inserted into JPMS 602 may change that value to ‘true,’ according to the policy enforced by policy engine 608. Doing so allows the failure to be logged by MJM 604, while still allowing the application to execute without actually failing.

An example module policy that may be enforced by policy engine 608 is as follows:

```

module-policy-settings:
- module-name: "name" // Module requesting
  other-module-name: "name" // Optional: Other Module (may be absent)
  module-operation: "operation" // Request type: open/read/export
  module-packages: ["package","package",etc.] // Optional: List of packages
  module-loader-name: "classloadername" // Optional: Class Loader associated
  with Module
  module-action: allow // Action: allow/default (either allow the operation or let
  the JVM decide)

```

During execution, MJM 604 may operate in conjunction with JPMS 602 to provide any or all of the following:

The ability to audit and ‘pass’ all module requests between Java modules 612 that would have ‘failed,’ which could be used in testing, scoping, etc.

The ability to script a module policy that goes far beyond what is provided by Java itself (e.g., the use of wildcards, regular expressions, etc.).

The ability to ‘build’ corrective command line arguments that can be used to correct module requests that have failed.

In other words, MJM 604 may effectively act as an agent for JPMS 602 that alters decisions made by JPMS 602. To do so, instrumenter 606 of MJM 604 may insert instrumentation 614 into the portion of JPMS 602 that checks the “linkage” between Java modules 612 in terms of their exports, reads, and opens. These checks are found the “Module” Class under the following methods:

TABLE 1

boolean	canRead(Module other)	Indicates whether this module reads the given module.
boolean	isExported(String pn)	Returns true if this module exports the given package unconditionally
boolean	isExported(String pn, Module other)	Returns true if this module exports the given package to at least the given module.
boolean	isOpen(String pn)	Returns true if this module has opened a package unconditionally.
boolean	isOpen(String pn, Module other)	Returns true if this module has opened a package to at least the given module.

When any of these methods are called within JPMS 602, the instrumentation 614 inserted by MJM 604 logs the Java modules 612 involved, the operation requested, the package (s) involved, and/or the decision made within JPMS 602 in terms of whether or not the module has the permission to successfully open, export, and/or read (e.g., the three key permissions for modules).

In various embodiments, the module policy may also support the use of wildcards and/or expressions. For example, if the policy field is omitted, it may be the same as “*”. Also, the module policy may also support the specification of “unnamed modules” using, e.g., the module name “Unnamed@ClassLoaderName”.

For example, the following “enableall.policy” will enable all module operations for purposes of testing and assessing the application:

```

module-policy-settings:
  module-name: "*"
  other-module-name: "*"
  module-operation: "*"
  module-packages: "*"
  module-loader-name: "*"
  module-action: allow

```

Once MJM 604 has logged information regarding Java modules 612, their open check calls, read check calls, export check calls, cached check calls, etc., MJM 604 may provide such information to the user via UI 610 (e.g., as one or more reports). In various embodiments, MJM 604 may be further configured to provide any command line fixes to UI 610 for any of the failures detected. This allows the user to quickly and easily address these fixes within Java modules 612 of the application.

A prototype system was implemented, to demonstrate the efficacy of the techniques herein. To do so, the MJM was built using the following build command:

```
gradlew clean build
```

In terms of compilation, the compiler (javac) also implements the module system and was setup with the MJM. However, the -javaagent switch cannot be used in this case and the use of the JAVA_TOOL_OPTIONS env symbol is recommended, as follows:

```
set JAVA_TOOL_OPTIONS=javaagent:prod/lib/
  javaagent.jar=agentConfig.yml
```

Then, the application can be run by attaching the javaagent to the application as follows:

-javaagent:prod/lib/javaagent.jar=agentConfig.yml
The core configuration file, ModularHandler.properties, was set as follows:

```
operation.cache.size=100 // how many operations to cache for display
module.policy.file=enableall.policy //which policy to use
```

As a simple test, the demo application included the method call shown below, which accesses the java.sql module. By default, it would have successfully read and exported. However, it would not be open to the module being passed in (e.g., a named module), so the reflection call would fail without the MJM.

```
private static void testSQL(Module module1) throws Exception
{
Enumeration<Driver> drivers=java.sql.DriverManager.getDrivers( );
System.out.println("Drivers are "+drivers);
Module sqlModule=java.sql.DriverManager.class.getModule( );
System.out.println("Drivers module name is "+sqlModule);
boolean isExported=sqlModule.isExported("java.sql",module1);
boolean canRead=module1.canRead(sqlModule);
boolean isOpen=sqlModule.isOpen("java.sql",module1);
System.out.println("SQL Module Permissions: "+isExported+" "+canRead+"
"+isOpen);
Field f=java.sql.DriverManager.class.getDeclaredField("logWriter");
f.setAccessible(true);
System.out.println("Value for logWriter is "+f.get(null));
}
```

FIGS. 7A-7B illustrate example screen captures of the execution of a test application with and without the use of an MJM in accordance with one or more embodiments described herein. As shown in FIG. 7A, screen capture 700 illustrates the execution of the test code above without the prototype MJM. From this, it can be seen that open is 'false' in the permissions for the java.sql module, resulting in the reflection call failing.

Conversely, screen capture 710 illustrates the execution of the same test code above with the prototype MJM. Here, the MJM has altered the permission for open to be 'true' in the permissions for the java.sql module, in accordance with its module policy. As a result, the reflection call succeeds. This allows the application to continue to execute, while still allowing the MJM to record information about the failure. Doing so provides the user with a series of diagnostics that can be leveraged to easily correct the application for Java 9+ compliance.

FIGS. 8A-8F illustrate example screen captures of the outputs of a prototype MJM system in accordance with one or more embodiments described herein. As noted above, the MJM may output various diagnostics captured during the execution of the application as one or more reports. Accordingly, the prototype MJM used a local, built-in web server to display a help screen with the following options:

```
Show Modules
Show Open Check Calls
Show Read Check Calls
Show Export Check Calls
Show Cached Checks
Print Module Command Line Fixes
```

FIG. 8A illustrates a screen capture 800 of the prototype system when the 'Show Modules' option is selected. This option shows all of the modules executed by the application and their information. For example, the resulting report may include various module information like its name, layer within the application, loader, opens (if applicable), required info, etc.

FIG. 8B illustrates a screen capture 810 of the prototype system when the "Show Open Check Calls" option is selected. This option shows all open check calls in a summarized manner. For example, the resulting report may list a summary of all module access checks for 'open' operations and indicate the opening module, the module to which it's opening, the package name, the number of calls, the results of the opens, and whether any override was made via policy by the MJM.

FIG. 8C illustrates a screen capture 820 of the prototype system when the "Show Read Check Calls" option is selected. This option shows all read check calls in summarized manner. For example, the resulting report may list a summary of all module access checks for 'read' operations

and indicate the reading module, the module that it can read, the package name, the number of calls, the results of the reads, and whether any override was made via policy by the MJM.

FIG. 8D illustrates a screen capture 830 of the prototype system when the "Show Export Check Calls" option is selected. This option shows all export check calls in a summarized manner. For example, the resulting report may list a summary of all module access checks for 'export' operations and indicate the exporting module, the module to which it is exported, the package name, the number of calls, the results of the exports, and whether any override was made via policy by the MJM.

FIG. 8E illustrates a screen capture 840 of the prototype system when the "Show Cached Checks" option is selected. This option shows all cached check instances in a summarized manner. For example, the resulting report may, for a given cached module access check, indicate a timestamp, type (e.g., isExported, canRead, etc.), module, other module, package name, result, etc.

FIG. 8F illustrates a screen capture 850 of the prototype system when the "Print Module Command Line Fixes" option is selected. This option shows the correct command line policies that can be added to the Java command line, to apply the module policy.

In closing, FIG. 9 illustrates an example simplified procedure for assessing Java 9+ compliance in accordance with one or more embodiments described herein. For example, a non-generic, specifically configured device (e.g., device 200) may perform procedure 900 by executing stored instructions (e.g., process 248, such as an "agent" process). The procedure 900 may start at step 905, and continues to step 910, where, as described in greater detail above, the agent may insert instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed.

At step **915**, as detailed above, the agent may capture, using the instrumentation, an access check failure for an inter-module operation between the Java modules. For example, such an inter-module operation may comprise a read, open, export, or cache action attempted by one of the modules with respect to another. In such a case, the instrumentation may capture information about the operation such as the modules involved, the operation itself, the package(s) involved, and the like.

At step **920**, the agent may override, using the instrumentation, the access check failure, as described in greater detail above. For example, the instrumentation may override an output of a `canRead` method, an `isExported` method, or an `isOpen` method of a Module Class used in the Java Platform Module System. In various embodiments, the agent may override the access check failure according to a predefined policy that specifies whether a particular operation should be overridden on failure.

At step **925**, as detailed above, the agent may report the captured access check failure to a user interface. For example, the agent may provide any or all of the information captured by the instrumentation about the access check failure to the user interface such as the modules involved, the failed inter-module operation, the package involved, etc.

The simplified procedure **900** may then end in step **935**, notably with the ability to continue ingesting and assessing data. Other steps may also be included generally within procedure **900**.

It should be noted that while certain steps within procedure **900** may be optional as described above, the steps shown in FIG. **9** are merely examples for illustration, and certain other steps may be included or excluded as desired. Further, while a particular order of the steps is shown, this ordering is merely illustrative, and any suitable arrangement of the steps may be utilized without departing from the scope of the embodiments herein.

The techniques described herein, therefore, provide for a modular Java management agent that can aid application developers and stakeholders migrate applications to Java+ and provides important features throughout the entire application lifecycle. In various aspects, the MJM implements an easier to use system for module configuration using regex, etc., instead of what is provided in base Java. In addition, the MJM includes a test mode that allows all issues to be detected at one, without having to repeatedly test the application and identify a failure. Further, the MJM provides a diagnostic panel that allows the user to review the inter-module calls and any failures that would have occurred.

Indeed, the techniques herein can guarantee that module restrictions will not impact the application runtime. In addition, the techniques herein can apply a policy to the configuration of modules in a single file that allows the use of wildcards and regex expressions. Further, the techniques herein can autogenerate commands to add to the command line and provide a complete audit of all module operations (e.g., opens, exports, and reads).

Specifically, according to various embodiments, an agent inserts instrumentation into a Java Platform Module System in which a plurality of Java-based modules of an application is executed. The agent captures, using the instrumentation, an access check failure for an inter-module operation between the Java modules. The agent overrides, using the instrumentation, the access check failure. The agent reports the captured access check failure to a user interface.

Illustratively, the techniques described herein may be performed by hardware, software, and/or firmware, such as in accordance with the illustrative agent profiler process

248, which may include computer executable instructions executed by the processor **220** to perform functions relating to the techniques described herein, e.g., in conjunction with corresponding processes of other devices in the computer network as described herein (e.g., on network agents, controllers, computing devices, servers, etc.).

According to the embodiments herein, a method herein may comprise: inserting, by an agent, instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed; capturing, by the agent and using the instrumentation, an access check failure for an inter-module operation between the Java modules; overriding, by the agent and using the instrumentation, the access check failure; and reporting, by the agent, the captured access check failure to a user interface.

In one embodiment, the inter-module operation comprises at least one of: an open, read, export, or cache action. In another embodiment, overriding the access check failure comprises overriding an output of a `canRead` method, an `isExported` method, or an `isOpen` method of a Module Class used in the Java Platform Module System. In a further embodiment, reporting the captured access check failure to the user interface comprises sending report data to the user interface that is indicative of the module attempting the inter-module operation, a target module for the operation, and a package associated with the attempted operation. In yet another embodiment, the agent overrides the access check failure according to a predefined policy. In a further embodiment, the agent also provides data to the user interface that is indicative of one or more command line commands that would correct the access check failure. In another embodiment, the application continues to execute as a result of the access check failure being overridden.

According to the embodiments herein, a tangible, non-transitory, computer-readable medium herein may have computer-executable instructions stored thereon that, when executed by a processor on a computer, may cause an agent to perform a method comprising: inserting, by the agent, instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed; capturing, by the agent and using the instrumentation, an access check failure for an inter-module operation between the Java modules; overriding, by the agent and using the instrumentation, the access check failure; and reporting, by the agent, the captured access check failure to a user interface.

Further, according to the embodiments herein an apparatus herein may comprise: one or more network interfaces to communicate with a network; a processor coupled to the network interfaces and configured to execute one or more processes; and a memory configured to store a process executable by the processor, the process, when executed, configured to: insert instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed; capture, using the instrumentation, an access check failure for an inter-module operation between the Java modules; override, using the instrumentation, the access check failure; and report the captured access check failure to a user interface.

While there have been shown and described illustrative embodiments above, it is to be understood that various other adaptations and modifications may be made within the scope of the embodiments herein. For example, while certain embodiments are described herein with respect to certain types of networks in particular, the techniques are not limited as such and may be used with any computer network, generally, in other embodiments. Moreover, while specific

technologies, protocols, and associated devices have been shown, such as Java, TCP, IP, and so on, other suitable technologies, protocols, and associated devices may be used in accordance with the techniques described above. In addition, while certain devices are shown, and with certain functionality being performed on certain devices, other suitable devices and process locations may be used, accordingly. That is, the embodiments have been shown and described herein with relation to specific network configurations (orientations, topologies, protocols, terminology, processing locations, etc.). However, the embodiments in their broader sense are not as limited, and may, in fact, be used with other types of networks, protocols, and configurations.

Moreover, while the present disclosure contains many other specifics, these should not be construed as limitations on the scope of any embodiment or of what may be claimed, but rather as descriptions of features that may be specific to particular embodiments of particular embodiments. Certain features that are described in this document in the context of separate embodiments can also be implemented in combination in a single embodiment. Conversely, various features that are described in the context of a single embodiment can also be implemented in multiple embodiments separately or in any suitable sub-combination. Further, although features may be described above as acting in certain combinations and even initially claimed as such, one or more features from a claimed combination can in some cases be excised from the combination, and the claimed combination may be directed to a sub-combination or variation of a sub-combination.

For instance, while certain aspects of the present disclosure are described in terms of being performed “by a server” or “by a controller”, those skilled in the art will appreciate that agents of the application intelligence platform (e.g., application agents, network agents, language agents, etc.) may be considered to be extensions of the server (or controller) operation, and as such, any process step performed “by a server” need not be limited to local processing on a specific server device, unless otherwise specifically noted as such. Furthermore, while certain aspects are described as being performed “by an agent” or by particular types of agents (e.g., application agents, network agents, etc.), the techniques may be generally applied to any suitable software/hardware configuration (libraries, modules, etc.) as part of an apparatus or otherwise.

Similarly, while operations are depicted in the drawings in a particular order, this should not be understood as requiring that such operations be performed in the particular order shown or in sequential order, or that all illustrated operations be performed, to achieve desirable results. Moreover, the separation of various system components in the embodiments described in the present disclosure should not be understood as requiring such separation in all embodiments.

The foregoing description has been directed to specific embodiments. It will be apparent, however, that other variations and modifications may be made to the described embodiments, with the attainment of some or all of their advantages. For instance, it is expressly contemplated that the components and/or elements described herein can be implemented as software being stored on a tangible (non-transitory) computer-readable medium (e.g., disks/CDs/RAM/EEPROM/etc.) having program instructions executing on a computer, hardware, firmware, or a combination thereof. Accordingly, this description is to be taken only by way of example and not to otherwise limit the scope of the embodiments herein. Therefore, it is the object of the

appended claims to cover all such variations and modifications as come within the true intent and scope of the embodiments herein.

What is claimed is:

1. A method, comprising:
 - inserting, by an agent, instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed;
 - capturing, by the agent and using the instrumentation, an access check failure for an inter-module operation between the Java modules;
 - overriding, by the agent and using the instrumentation, the access check failure; and
 - reporting, by the agent, the captured access check failure to a user interface.
2. A method as in claim 1, wherein the inter-module operation comprises at least one of: an open, read, export, or cache action.
3. A method as in claim 1, wherein overriding the access check failure comprises:
 - overriding an output of a canRead method, an isExported method, or an isOpen method of a Module Class used in the Java Platform Module System.
4. A method as in claim 1, wherein reporting the captured access check failure to the user interface comprises:
 - sending report data to the user interface that is indicative of the module attempting the inter-module operation, a target module for the operation, and a package associated with the attempted operation.
5. A method as in claim 1, wherein the agent overrides the access check failure according to a predefined policy.
6. A method as in claim 1, further comprising:
 - providing data to the user interface that is indicative of one or more command line commands that would correct the access check failure.
7. A method as in claim 1, wherein the application continues to execute as a result of the access check failure being overridden.
8. A tangible, non-transitory, computer-readable medium having computer-executable instructions stored thereon that, when executed by a processor on a computer, cause an agent to perform a method comprising:
 - inserting, by the agent, instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed;
 - capturing, by the agent and using the instrumentation, an access check failure for an inter-module operation between the Java modules;
 - overriding, by the agent and using the instrumentation, the access check failure; and
 - reporting, by the agent, the captured access check failure to a user interface.
9. The computer-readable medium as in claim 8, wherein the inter-module operation comprises an open, read, export, or cache action.
10. The computer-readable medium as in claim 8, wherein overriding the access check failure comprises:
 - overriding an output of a canRead method, an isExported method, or an isOpen method of a Module Class used in the Java Platform Module System.
11. The computer-readable medium as in claim 8, wherein reporting the captured access check failure to the user interface comprises:
 - sending report data to the user interface that is indicative of the module attempting the inter-module operation, a target module for the operation, and a package associated with the attempted operation.

25

12. The computer-readable medium as in claim 8, wherein the agent overrides the access check failure according to a predefined policy.

13. The computer-readable medium as in claim 8, wherein the method further comprises:

providing data to the user interface that is indicative of one or more command line commands that would correct the access check failure.

14. The computer-readable medium as in claim 8, wherein the application continues to execute as a result of the access check failure being overridden.

15. An apparatus, comprising:

one or more network interfaces to communicate with a network;

a processor coupled to the network interfaces and configured to execute one or more processes; and

a memory configured to store a process executable by the processor, the process, when executed, configured to: insert instrumentation into a Java Platform Module System in which a plurality of Java modules of an application is executed;

capture, using the instrumentation, an access check failure for an inter-module operation between the Java modules;

override, using the instrumentation, the access check failure; and

26

report the captured access check failure to a user interface.

16. The apparatus as in claim 15, wherein the inter-module operation comprises at least one of: an open, read, export, or cache action.

17. The apparatus as in claim 15, wherein the apparatus overrides the access check failure by:

overriding an output of a canRead method, an isExported method, or an isOpen method of a Module Class used in the Java Platform Module System.

18. The apparatus as in claim 15, wherein the apparatus reports the captured access check failure to the user interface by:

sending report data to the user interface that is indicative of the module attempting the inter-module operation, a target module for the operation, and a package associated with the attempted operation.

19. A apparatus as in claim 15, wherein the apparatus overrides the access check failure according to a predefined policy.

20. The apparatus as in claim 15, wherein the process when executed is further configured to:

provide data to the user interface that is indicative of one or more command line commands that would correct the access check failure.

* * * * *

UNITED STATES PATENT AND TRADEMARK OFFICE
CERTIFICATE OF CORRECTION


PATENT NO. : 11,436,030 B2
APPLICATION NO. : 16/788041
DATED : September 6, 2022
INVENTOR(S) : Walter Theodore Hulick, Jr.

Page 1 of 1

It is certified that error appears in the above-identified patent and that said Letters Patent is hereby corrected as shown below:

In the Specification

Column 11, Line 48, please amend as shown:
one or more separate machines. When network 420 is the

Signed and Sealed this
Sixth Day of December, 2022

Katherine Kelly Vidal
Director of the United States Patent and Trademark Office